The BetaDBC Library

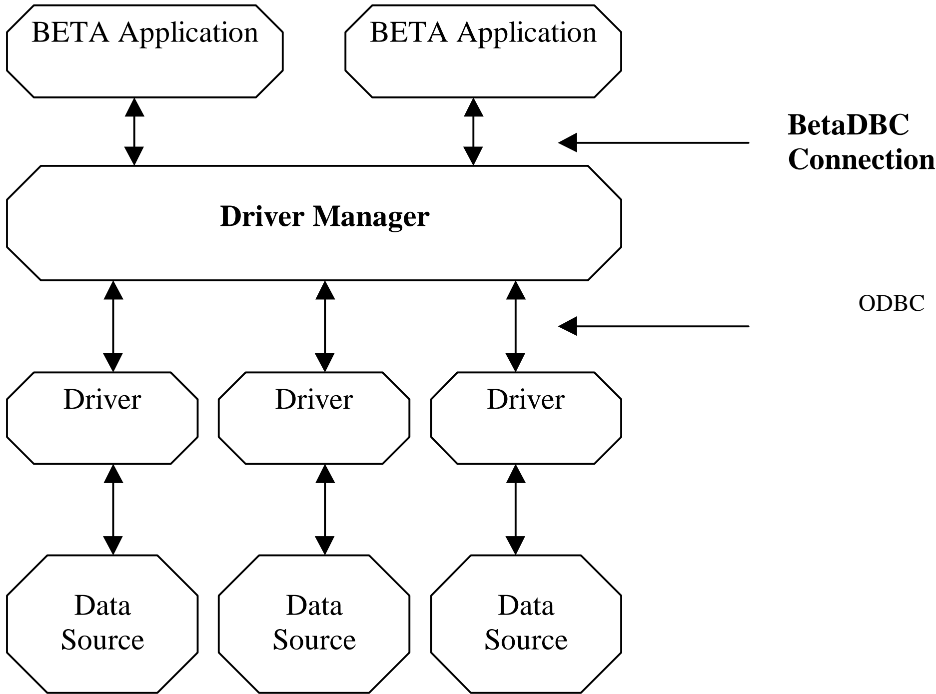
This document describes the BetaDBC (Beta DataBase Connectivity) library for communicating with relational databases using SQL (Structured Query Language, [Date 1993]). The library implements the pattern `connection` used to model connections to relational databases. This pattern contains patterns for querying and manipulating relational databases. Additional support for transactions may be added by including the fragment `transactions`.

1 BetaDBC Basics

Generally speaking there are two ways of programming to a relational database, namely using

1. *Embedded SQL*. Using this approach SQL statements are embedded in a host language. Using a Database Management System (DBMS) specific precompiler, the application program is precompiled, transforming statements containing *shared variables* (variables shared by the DBMS and the application program) into DBMS library calls. After compilation the program is linked with the DBMS' runtime library.
2. *Call Level Interfaces (CLIs)*. These are libraries of functions. The functions are the native application programming interface of the DBMS or calls to it. In this way, a precompiler is not needed.

Both approaches suffer from some problems although standards have been proposed: Embedded SQL ties a program to a specific DBMS meaning that at least a new compilation will have to be made if a new DBMS is to be used. CLIs are hard to learn and often contain a lot of DBMS specific functions.
BetaDBC combines these two approaches in such a way that although shared variables may be used a precompiler is not needed and furthermore all DBMS may be treated alike. In order to achieve this BetaDBC currently builds upon and extends Open DataBase Connectivity (ODBC [Geiger, 1995]) – an industry standard for CLIs.
This means that the architecture of a typical BetaDBC application may be outlined as below:



A BETA application typically connects (via a connection) to a DBMS via BetaDBC, calls BetaDBC functions, processes results and disconnects. The driver manager loads and unloads drivers as requested by applications, and processes function calls before sending them to a driver. The driver then processes the functions calls, submits SQL requests to data sources and returns results. Data sources encapsulate the data in form of tables that a user wants to access together with an associated operating system, DBMS, and network platform (if applicable) used to access the DBMS.
Details on how to create data sources and use BetaDBC in this environment will be given below.

2 The BetaDBC Interface

The BetaDBC interface is built around the concept of a `connection` that models a connection to a relational DBMS. The BetaDBC interface to a `connection` is (see below for the full interface):

```
Connection: (*)
  (# <<SLOT ConnectionLib:Attributes>>;
     declareVar: (*) ...;
     declareInteger: (*) declareVar ...;
     declareReal: (*) declareVar ...;
     declareText: (*) declareVar ...;
     declareBoolean: (*) declareVar ...;
     declareDate: (*) declareVar ...;
     declareTime: (*) declareVar ...;
     SQLStatement: (*) ...;
     directSQLStatement: (*) SQLStatement ...;
     preparedSQLStatement: (*) directSQLStatement ...;
     resultSet: (*) ...;
     open:< (*) ...;
     close:< ...;
     connectionException:< BDBCException ...;
     connectionWarning:< BDBCWarning ...;
     private: @<<SLOT ConnectionPrivate:Descriptor>>
  #);
```

2.1 Data Sources

In order to use a `connection` a data source must be created (See the tutorial for specifics.). To communicate with an existing data source the user must first create an instance of the `connection` pattern. Calling the `open` method on a `connection`:

```
open:< (*)
  (# name: ^text;
     userName: ^text;
     password: ^text;
     openConnectionException:< BDBCException (#  do INNER #);
     openConnectionWarning:< BDBCWarning (#  do INNER #)
  enter (name[],userName[],password[])
  <<SLOT ConnectionOpen:DoPart>>
  #);
```

then makes it possible to communicate with the data source. When calling open the name of the data source must be supplied whereas user name and/or password may be omitted as appropriate. Consider as an example the statement

```
('ullman97','marius',none)->sqlCon.open
```

The statement opens a connection to the data source name 'ullman97' for the user 'marius' without specifying a password.

2.2 Shared variables

Definition of shared variables are done via the '`declare`…' methods. To e.g. declare a text `txt` as a shared variable named 'txt' use

```
'studioName'->declareText
  (# set::(# do value->studioName #);
     get::(# do studioName[]->value[] #)
  #)
```

2.3 SQL Statements

Data manipulation and definition is done using the SQL statement patterns. The SQL statement patterns have the following interfaces:

```
SQLStatement:(*)
  (# <<SLOT SQLStatementLib:Attributes>>;
     execute:<(*) ...;
     close:(*) ...;
     execException:< BetaDBCException ...;
     execWarning:< BetaDBCWarning ...;
     private: @<<SLOT SQLStatementPrivate:Descriptor>>;
     get:< ...;
     set:< ...
  enter set
  do INNER
  exit get
  #);

directSQLStatement:(*) SQLStatement
  (# <<SLOT DirectSQLStatementLib:Attributes>>;
     currentMarker:(*) @ ...;
     marker:(*) ...;
     b: (*) marker ...;
     c: (*) marker ...;
     d: (*) marker ...;
     f: (*) marker ...;
     i: (*) marker ...;
     s: (*) marker ...;
     t: (*) marker ...;
     execute::  ...;
     execDirectException:< BetaDBCException ...;
     execDirectWarning:< BetaDBCWarning ...;
     private: @<<SLOT DirectSQLStatementPrivate:Descriptor>>;
     set::<  ...;
     getExpanded:(*) ...
  do INNER
  #);

preparedSQLStatement:(*) ...
```

To use a statement `stmt` one must first associate it with an SQL statement as in

```
'SELECT title, length FROM Movie WHERE studioName = ''Disney'''->stmt
```

Invoking `execute` on `stmt` will then cause the SQL statement to be executed at the database.
A `preparedSQLStatement` differs from a `directSQLStatement` in that a prepared statement is parsed and prepared by the data source when the statement is initialised, i.e. executing the statement above will, if `stmt` is a `preparedSQLStatement`, cause the contents of the SQL statement to be sent to the database in order for it to be parsed and prepared for future execution. If `stmt` is a `directSQLStatement` no communication with the database will occur before calling `execute`.
In this way a prepared SQL statement is a little slower to initialise than a direct SQL statement but much faster to execute. Use a `preparedSQLStatement` only when an SQL statement has to be executed several times.[1]

The contents of a `directSQLStatement` can be any SQL statement with embedded shared variables and/or markers, i.e. a `directSQLStatement stmt` may be initialised as in

```
'SELECT title,length FROM Movie WHERE studioName = :studioName AND year = %i'
  ->stmt
```

Here 'studioName' is the name of a shared variable declared as shown above. The value of the `%i` marker may be set using the `i` pattern in `directSQLStatement`. Now suppose that the following statements have been executed

```
do ...; 'Disney'->studioName; 1990->stmt.i; ...
```

When executing the SQL statement the SQL contents of the statement will then conceptually be

```
SELECT title,length FROM Movie WHERE studioName = 'Disney' AND year = 1990
```

i.e. before sending an SQL statement to a database the embedded shared variables and the markers are, conceptually, substituted for their current values. After execution the contents of the statement can be changed, the markers can be reset or the statement can be closed.
A preparedSQLStatement is used similarly to a directSQLStatement.

2.4 Results

Executing an SQL statement `stmt` will yield an instance of `resultSet` (here `rs` is a reference to a resultSet):

```
stmt.execute->rs[]
```

A `resultSet` implements an interface to an SQL cursor in the following way

[1] Note that, currently, a `preparedSQLStatement` is implemented as a `directSQLStatement`.

```
resultSet:(*)
  (# <<SLOT ResultSetLib:Attributes>>;
     columnCount:(*) integerValue ...;
     rowCount:(*) integerValue ...;
     column: (*)...;
     getColumn: (*)...;
     getColumnByName:(*) ...;
     result:(*)
       (# <<SLOT ResultLib:Attributes>>;
          marker:(*) ...;
          b: marker ...;
          c: marker ...;
          d: marker ...;
          f: marker ...;
          i: marker ...;
          s: marker ...;
          t: marker ...;
          private: @...
       #);
     scan:(*)
       (# current: @result;
          varNotDeclared:<(*) exception ...;
          unknownColumn:<(*) exception ...;
          pattern: ^text
       enter pattern[]
       ...
       #);
     resultSetException:< BDBCException ...;
     resultSetWarning:< BDBCWarning ...;
     private: @...
  #)
```

Given a `resultSet` the `scan` method iterates over the `results` in the `resultSet`. There are three distinct ways to control the scan. Firstly, one may simply execute

```
rs.scan(# ... #)
```

In the do-part of the scan one may then refer to the values of the columns in the result. This is done sequentially by referring to the markers of the current `result`. If e.g. `rs` was retrieved as shown above,

```
rs.scan(# do current.s -> puttext; put; current.i->putint; newline #)
```

will scan over the results in the `resultSet` and print the values of their columns on the screen. Also, one may enter a string when evaluating a scan pattern as in:

```
':title %i'->rs.scan(# do title->puttext; put; current.i->putint; newline #)
```

Here `title` is a shared variable named 'title'. This statement prints the same as above but by providing an input string it is here specified that the first column of each `result` should be assigned to the shared variable 'title' and that the second column of each `result` is an integer that will be fetched via the `i` marker. In general one may in this way specify how each column of a `result` should be treated.

The two ways of scanning shown above may, in some circumstances, be problematic in that they assume a specific ordering of columns in the `results`. Therefore, the last way of doing a scan names the columns in the `resultSet`, as in e.g.:

```
'length:%i title:title'->rs.scan
   (# do title->puttext; put; current.i->putint; newline #)
```

In this way the order of the columns in the result may be changed from 'title,length' to 'length,title' without any problems for the last way of scanning.

3 Transactions

Many data sources support the use of transactions. In order to use this capability from BetaDBC the fragmen 'transactions' may be included. An outline of the interface of this fragment is shown below. The full interface may be seen in section 5.3.

```
ORIGIN 'betadbc';
BODY 'private/transactionsbody';
-- connectionLib: Attributes --
transactionsSupported:(*)
  (# isSupported: @boolean
  ...
  exit isSupported
  #);
autoCommitMode:(*)
  (# autoCommit: @boolean
  enter (#  enter autoCommit ... #)
  exit
    (# ...
    exit autoCommit
    #)
  #);
readUncommitted: (*) integerValue (# ... #);
readCommitted: (*) integerValue (# ... #);
repeatableRead: (*) integerValue (# ... #);
serializable:(*) integerValue (# ... #);
transactionLevelSupported:(*) booleanValue
  (# level: (*) @integer
  enter level
  ...
  #);
transactionLevel:(*)
  (# level: (*) @integer
  enter (#  enter level ... #)
  exit
    (# ...
    exit level
    #)
  #);
commit:(*) (#  ... #);
rollBack:(*) (#  ... #)
```

The scope of a transaction is a whole connection including all statements allocated in it. The default is that every execution of an SQL statement start a new transaction and automatically commits the effects of this statement after the statement has completed. This autocommit mode may be chagned to manual commit mode by by evaluating

```
false->sqlCon.autoCommitMode
```

where `sqlCon` is an instance of a `connection`. Note that this is only meaningful if `transactionsSupported` evaluates to `true`. In manual commit mode a series of database manipulations may be committed by executing `commit`. Equivalently a series of database manipulations may be aborted by executing `rollBack`.
Four transaction isolation levels (as defined in the SQL standard) are available, namely `readUncommitted`, `readCommitted`, `repeatableRead`, and `serializable`. Whether a transaction isolation level, such as `serializable`, is supported in a given connection may be checked by evaluating e.g.

```
serializable->sqlCon.transactionLevelSupported
```

4 Tutorial

The sample programs shown in the tutorial may be found in the `tutorial` directory accompanying BetaDBC. The examples use a database schema and examples from [Ullman, 1997]. It supposes that the reader is familiar with basic SQL and focuses on teaching the essentials of using BetaDBC.

4.1 Creating a data source

In order to use BetaDBC a suitable data source must be created. Currently, data sources are created outside BetaDBC. Different procedures must be followed depending on the operating system used. This will be changed in a future release of BetaDBC.

4.1.1 Creating a data source on Windows 95 and NT

In the ”Start” menu choose ”Settings” and ”Control Panel”. Start the ODBC (32 Bit) application from the “Control Panel”. Press ”Add…” and choose the database driver that you want to use, then press ”Finish”. Follow the driver specific instructions.

4.1.2 Creating a data source on Unix

You will need a `.odbc.ini` file in your home directory. Create such a file if it does not exist. A sample `.odbc.ini` file may look like

```
[ullman97]
# movie database in PostgreSQL
Driver = /users/postgres/lib/libcliPG.so
...

[default]
# default to odbc gateway
Driver = /users/postgres/lib/libcliPG.so
```

Here ‘ullman97’ and ‘default’ are names of data sources. Each data source at least specifies which ODBC driver should be used when connecting to that data source, both data sources above a driver for the PostgreSQL database is used. Furthermore it may be necessary to specify some driver specific attributes of the data source. This is tentatively done with the ‘...’s above. See the specific driver’s documentation for details. << Lidt bedre forklaring her når PostgreSQL er blevet sat rigtigt op>>.

In the following it is assumed that a suitable data source named “ullman97” has been created.

4.2 Creating a database

The next step will then be to create and insert values into a database. Let’s use the following sample database schema

```
Movie(title, year, length, inColor, studioName, producerCNo)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name,address,gender,birthdate)
MovieExec(name, address, certNo, netWorth)
Studio(name, address, presCNo)
```

A series of SQL statements creating the database schema may be found in `createmoviedbtables.txt`. The text files used in this section are shown in section 6.
Create the tables corresponding to this schema by running the `executesqlfile` program also found in the `tutorials` directory:

```
[postgres@delirium tutorial]$ ./executesqlfile createmovie.txt ullman97
```

How the `executesqlfile` program is implemented will be discussed later. You may now insert some values in the database by running

```
[postgres@delirium tutorial]$ ./executesqlfile moviedbtables.txt ullman97
```

The tables may later be deleted by running

```
[postgres@delirium tutorial]$ ./executesqlfile dropmovies.txt ullman97
```

4.3 Querying and retrieving from the database

This section will introduce the basics of BetaDBC: connecting to data sources, executing simple queries and retrieving the results.
Consider the simple SQL statement

```
SELECT *
FROM Movie
WHERE studioName = ‘Disney’ AND year = 1990;
```

An application that uses BetaDBC, executes the above query and retrieves the result may look like:

```
ORIGIN '../betadbc';
-- program: Descriptor --
(#
   sqlCon: @connection;
   stmt: @sqlCon.directSqlStatement;
   rs: ^sqlCon.resultSet
do
   (2->arguments,3->arguments,4->arguments)->sqlCon.open;
   'SELECT title, length FROM Movie WHERE studioName  = ''Disney'' AND year = 1990'
     ->stmt;
   stmt.execute->rs[];
   rs.scan
     (#
     do
        'title: '->puttext;
        current.s->putline;
        'length: '->puttext;
        current.i->putint;
        newline
     #);
   stmt.close
   sqlCon.close
#)
```

The program starts out by declaring a connection, a `directSQLStatement` belonging to that connection and a `resultSet` belonging to that connection. The connection is used in order to connect to a data source in the first line of the program:

```
(2->arguments,3->arguments,4->arguments)->sqlCon.open;
```

`Connections` `open` method takes as arguments a name of the `connection`, a username and a password. Thus an invocation of the program like

```
[postgres@delirium tutorial]$ ./simple ullman97 marius foobar
```

means that the first statement will be an attempt to connect the user “marius” with password “foobar” to the data source named ‘ullman97’. If this succeeds

```
'SELECT title, length FROM Movie WHERE studioName = ''Disney'' AND year = 1990'
     ->stmt;
```

will set the contents of the `directSQLStatement` stmt to the query we want to execute. Executing the query yields a `resultSet` holding a cursor for the result

```
stmt.execute->rs[];
```

The `resultSet` may then be scanned. During the scan “`current`” will hold a reference to a result representing a row in the `resultSet`. The values of this result may then be accessed consecutively by using the marker attributes

```
rs.scan
    (#
    do
       'title: '->puttext;
       current.s->putline;
       'length: '->puttext;
       current.i->putint;
       newline
    #);
```

Finally, in order to free resources the `directSQLStatement` and the `connection` are closed.

4.4 Executesqlfile

The simple scheme presented in the last section can now be used for implementing the `executesqlfile` program. The `executeLoop` shows how to reuse an `SQLStatement` by simply replacing it's textual contents

```
executeTxt[]->sqlCon.stmt
```

4.5 Embedded SQL – Using Shared Variables

Using shared variables makes it possible to use the values of BETA objects in place of a concrete value in SQL statements. Using BETA and BetaDBC does not include using a preprocessor, which makes it necessary to declare shared variables imperatively, as in

```
   sqlCon:@connection
   studioName:@text;
do ...;
   'studioName'
     ->sqlCon.declareText
       (# set:: (#  do value->studioName #);
          get:: (#  do studioName[]->value[] #)
       #);
   ...
```

Here a shared text variable named 'studioName' is declared. The '`set`' pattern is final bound to describe how the shared variable's value is to be set. '`Get`' is final bound to describe how the value of the shared variable is to be fetched.
Then, using embedded SQL syntax, one may use shared variables in SQL statements:

```
   stmt:@sqlCon.directSQLStatement;
do ...;
   'INSERT INTO Studio(name, address) VALUES (:studioName, :studioAddr)'
     ->stmt;
   ...
```

This means that when executing `stmt`, '`:studioName`' and '`:studioAddr`' will (conceptually) be replaced by the values of the BETA textvariables '`studioName`' and '`:studioAddr`', and the resulting SQL statement will then be executed.
Using BetaDBC it is possible to declare most commonly used objects as shared variables (i.e. boolean, integer, real, text, date and time). The figure below shows a full program that will execute the statement above. '`stmt.getExpanded`' returns in a text how the SQL statement would look if it was executed at that point.

```
ORIGIN '../betadbc';
-- program: Descriptor --
(# sqlCon: @connection;
   stmt: @sqlCon.directSqlStatement;
   studioName,studioAddr: @text
do (2->arguments,3->arguments,4->arguments)->sqlCon.open;
   'studioName'
     ->sqlCon.declareText
       (# set:: (#  do value->studioName #);
          get:: (#  do studioName[]->value[] #)
       #);
   'studioaddr'
     ->sqlCon.declareText
       (# set:: (#  do value->studioAddr #);
          get:: (#  do studioAddr[]->value[] #)
       #);
   'Input a studio name: '->puttext ;
   getLine->studioName.puttext;
   ' and address: '->puttext;
   getLine->studioAddr.puttext;
   'INSERT INTO Studio(name, address) VALUES (:studioName, :studioAddr)'
     ->stmt;
   stmt.getExpanded->putline;
   stmt.close;
   sqlCon.close
#)
```

Program 3: embedded.bet

4.6 Embedded SQL – Fetching results

Suppose that we are executing a statement that return a result – as e.g. a select statement. Embedded SQL can then also be used to fetch results directly into shared variables. In BetaDBC this is done through the use of the scan pattern. Suppose we are executing

```
'SELECT MovieExec.name, netWorth FROM Studio, MovieExec WHERE presCNo = certNo
AND Studio.name = :studioName'
```

Then,

```
':presName :presNetWorth'
    ->(stmt.execute).scan
      (# ... #)
```

will cause the first column in each result tuple to be assigned to the shared integer variable '`presName`', and the second column to '`presNetWorth`'. The full code is shown below:

```
ORIGIN '../betadbc';
INCLUDE '~beta/basiclib/formatio';
-- program: Descriptor --
(# sqlCon: @connection;
   stmt: @sqlCon.directSqlStatement;
   studioName,presName: @text;
   presNetWorth: @integer
do 'studioName'
     ->sqlCon.declareText
       (# set::  (#  do value->studioName #);
          get::  (#  do studioName[]->value[] #)
       #);
   'presName'
     ->sqlCon.declareText
       (# set::  (#  do value->presName #);
          get::  (#  do presName[]->value[] #)
       #);
   'presNetWorth'
     ->sqlCon.declareInteger
       (# set::  (#  do value->presNetWorth #);
          get::  (#  do presNetWorth->value #)
       #);
   (2->arguments,3->arguments,4->arguments)->sqlCon.open;
   'Input a studio name: '->puttext;
   getLine->studioName.puttext;
   'SELECT MovieExec.name, netWorth FROM Studio, MovieExec WHERE presCNo =
certNo AND Studio.name = :studioName'
     ->stmt;
   stmt.getExpanded->putline;
   ':presName :presNetWorth'
     ->(stmt.execute).scan
       (#
       do 'The net worth of the president %s \nof %s is %i $\n'
            ->putFormat
              (#  do presName[]->s; studioName[]->s; presNetWorth->i #)
       #);
   stmt.close;
   sqlCon.close
#)
```

4.7 An adhoc query evaluator

We now have most of the building blocks to create an adhoc query evaluator, i.e. a program that connects to a data source and in a loop prompts for SQL statements that are to be executed on this data source. The following implements such a program.
As long as the user inputs anything but an empty line this input is sent to the data source as an SQL statement:

```
getline->stmt;
(if (stmt).empty then leave L if);
stmt.execute->res[];
```

If succesful, the result is examined. First the column information of the `resultSet` is extracted:

```
(for j: res.columnCount repeat
  '%s: %s\t'->putFormat
              (#
              do (j->res.getColumn).name[]->s;
                 (j->res.getColumn).dataTypeName[]->s
              #)
for);
```

Each resultSet has a `columnCount` yielding the number of columns in the `resultSet`. For each column, information such as `name` and `dataTypeName` (a DBMS specific datatype name) may be retrieved.
If the `columnCount` is non-zero, the results are fetched:

```
(if res.columnCount > 0 then
    res.scan
      (#
      do (for i: res.columnCount repeat
              (if (i->res.getColumn).DataType##
               // text## then
                  current.s->puttext
               // integerObject## then
                  current.i->putint
               // realObject## then
                  current.f->putreal
               // booleanObject## then
                  (if current.b then
                      'true'->puttext;
                   else
                      'false'->puttext
                  if)
               // time## then
                  current.t->puttime
               else
                  'Unknown data type!!!'->puttext
              if);
              '\t'->puttext
         for);
         newline
      #)
   else
      'DML/DDL statement executed successfully!'->putLine
if)
```

Again the information about columns in the `resultSet` is used. By evaluating

```
(i->res.getColumn).DataType##
```

the BETA pattern corresponding to the SQL datatype in column i is found.

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE '../betadbc'
        '~beta/basiclib/formatio'
        '~beta/basiclib/numberio';
-- program: Descriptor --
(# sqlCon: @Connection;
   stmt: @sqlCon.DirectSQLStatement;
   res: ^sqlCon.ResultSet
do (2->arguments,3->arguments,4->arguments)->sqlCon.open;
   L: cycle
     (#
     do 'Statement to execute: '->puttext;
        getline->stmt;
        (if (stmt).empty then leave L if);
        stmt.execute->res[];
        (for j: res.columnCount repeat
          '%s: %s\t'
            ->putFormat
              (#
              do (j->res.getColumn).name[]->s;
                 (j->res.getColumn).dataTypeName[]->s
              #)
        for);
        newline;
        (if res.columnCount > 0 then
            res.scan
              (#
              do (for i: res.columnCount repeat
                      (if (i->res.getColumn).DataType##
                       // text## then
                          current.s->puttext
                       // integerObject## then
                          current.i->putint
                       // realObject## then
                          current.f->putreal
                       // booleanObject## then
                          (if current.b then
                              'true'->puttext;
                           else
                              'false'->puttext
                          if)
                       // time## then
                          current.t->puttime
                       else
                          'Unknown data type!!!'->puttext
                      if);
                      '\t'->puttext
                 for);
                 newline
              #)
         else
            'DML/DDL statement executed successfully!'->putLine
        if)
     #);
   stmt.close;
```

Program 5: adhoc.bet

5 Interface Descriptions

5.1 Overview of ‘betadbc.bet’

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE '~beta/basiclib/timedate'
        'shortintobject';
BODY 'private/betadbcbody';
-- lib: Attributes --
Connection: (*)
  (#
     <<SLOT ConnectionLib:Attributes>>;
     declareVar: (*) ...;
     declareInteger: (*) declareVar ...;
     declareReal: (*) declareVar ...;
     declareText: (*) declareVar ...;
     declareBoolean: (*) declareVar ...;
     declareDate: (*) declareVar ...;
     declareTime: (*) declareVar ...;
     SQLStatement: (*) ...;
     directSQLStatement: (*) SQLStatement ...;
     preparedSQLStatement: (*) directSQLStatement ...;
     resultSet: (*) ...;
     open:< (*) ...;
     close:< ...;
     connectionException:< BDBCException ...;
     connectionWarning:< BDBCWarning ...;
     private: @<<SLOT ConnectionPrivate:Descriptor>>
  #);
BDBCException:(*) Exception ...;
BDBCWarning: BDBCException ...
```

5.2 Details of ‘betadbc.bet’

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE '~beta/basiclib/timedate';
BODY 'private/betadbcbody';
-- lib: Attributes --
Connection: (* A connection to a relational DBMS *)
  (#
     <<SLOT ConnectionLib:Attributes>>;
     declareVar:
     (* Used to declare a wrapper on a BETA object so that
      * object may be used in SQLStatements *)
       (#
          varName: ^text;
          set:<
          (* Furtherbind this to set the value of the declared
           * wrapper object to the value of the wrapped object *) object;
          get:<
          (* Furtherbind this to get the value of
           * the variable defined *) object
       enter varName[]
       <<SLOT ConnnectionDeclareVar:DoPart>>
       exit varName[]
       #);
     declareInteger:
     (* Declares a wrapper around an integer *) declareVar
       (#
          set::<  (# value: @integer enter value do INNER #);
          get::<  (# value: @integer do INNER exit value #)
       #);
     declareReal: (* Declares a wrapper around a real *) declareVar
```

```
  (#
     set::<  (# value: @real enter value do INNER #);
     get::<  (# value: @real do INNER exit value #)
  #);
declareText: (* Declares a wrapper around a text *) declareVar
  (#
     set::<  (# value: ^text enter value[] do INNER #);
     get::<  (# value: ^text do INNER exit value[] #)
  #);
declareBoolean: (* Declares a wrapper around a boolean *) declareVar
  (#
     set::<  (# value: @boolean enter value do INNER #);
     get::<  (# value: @boolean do INNER exit value #)
  #);
declareDate:
(* Declares a wrapper around a 'date'. Only the year, month and day
 * attributes of the time value object are taken into account
 *) declareVar
  (#
     set::<  (# value: @time enter value do INNER #);
     get::<  (# value: @time do INNER exit value #)
  #);
declareClock:
(* Declares a wrapper around a 'time'. Only the hour, minute and sec
 * attributes of the time value object are taken into account *)
 declareVar
  (#
     set::<  (# value: @time enter value do INNER #);
     get::<  (# value: @time do INNER exit value #)
  #);
declareTime: (* Declares a wrapper around a 'time' *) declareVar
  (#
     set::<  (# value: @time enter value do INNER #);
     get::<  (# value: @time do INNER exit value #)
  #);
SQLStatement:
(* The abstract superpattern for all SQL statements.
 * To use an instance of a subclass of SQLStatement:
 * 1. Set the statements SQL contents by evaluating the enter part
 * 2. Call 'execute'. Repeat 2, as necessary.
 * 3. Go to 1., if necessary
 * 4. Call 'close'
 *)
  (#
     <<SLOT SQLStatementLib:Attributes>>;
     execute:<
     (* Executes this(SQLStatement) *)
       (# res: ^ResultSet
       <<SLOT SQLStatementExecute:DoPart>>
       exit Res[]
       #);
     close:
     (* Closes this(SQLStatement).
      * Call close when done with this(SQLStatement) *)
     (#  <<SLOT SQLStatementClose:DoPart>> #);
     execException:< BetaDBCException
       (#  do INNER #);
     execWarning:< BetaDBCWarning
       (#  do INNER #);
     private:
       @<<SLOT SQLStatementPrivate:Descriptor>>;
     get:< (# t: ^text <<SLOT SQLStatementGet:DoPart>> exit t[] #);
```

```
     set:<
       (# t: ^text
       enter t[]
       <<SLOT SQLStatementSet:DoPart>>
       #)
  enter set
  do INNER
  exit get
  #);
directSQLStatement:
(* Use this statement type if a statement will be executed at most
 * a few times. The contents may contain placeholders in the form
 *   - :name designating that the shared variable named "name"
 *      will be bound to that place
 * In addition also
 * %b for booleans
 * %c for clocks
 * %d for dates
 * %f for reals
 * %i for integers
 * %s for texts
 * %t for time
 * may be used when setting the contents of this(directSQLStatement)
 *) SQLStatement
  (#
     <<SLOT DirectSQLStatementLib:Attributes>>;
     currentMarker:
     (* The current marker decides which placeholder will be set
      * if one of the subpatterns of "marker" is used
      *) @
       (#
          set:
            (#
            enter no
            <<SLOT DirectSQLStatementCurrentMarkerSet:DoPart>>
            #);
          get:
            (#
            <<SLOT directSQLStatemenCurrentMarkerGet:DoPart>>
            exit no
            #);
          no: @integer
       enter set
       exit get
       #);
     marker:
     (* Abstract superpattern for markers.
      * Advances currentMarker by 1
      *) (#  <<SLOT directSQLStatementMarker:DoPart>> #);
     b:
     (* Set placeholder number 'currentMarker'
      * to the boolean 'value' *) marker
       (# value: @boolean
       enter value
       <<SLOT directSQLStatementB:DoPart>>
       #);
     c:
     (* Set placeholder number 'currentMarker'
      * to the clock 'value' *) marker
       (# value: @time
       enter value
       <<SLOT directSQLStatementC:DoPart>>
```

```
      #);
    d:
    (* Set placeholder number 'currentMarker'
     * to the date 'value' *) marker
      (# value: @time
      enter value
      <<SLOT directSQLStatementD:DoPart>>
      #);
    f:
    (* Set placeholder number 'currentMarker'
     * to the real 'value' *) marker
      (# value: @real
      enter value
      <<SLOT directSQLStatementF:DoPart>>
      #);
    i:
    (* Set placeholder number 'currentMarker'
     * to the integer 'value' *) marker
      (# value: @integer
      enter value
      <<SLOT directSQLStatementI:DoPart>>
      #);
    s:
    (* Set placeholder number 'currentMarker'
     * to the text 'value' *) marker
      (# value: ^text
      enter value[]
      <<SLOT directSQLStatementS:DoPart>>
      #);
    t: (* Set placeholder number 'currentMarker'
        * to the time 'value' *) marker
      (# value: @time
      enter value
      <<SLOT directSQLStatementT:DoPart>>
      #);
    execute::
      (#
      <<SLOT DirectSQLStatementExecute:DoPart>>
      #);
    execDirectException:< BetaDBCException
      (#  do INNER execDirectException #);
    execDirectWarning:< BetaDBCWarning
      (#  do INNER execDirectWarning #);
    private: @<<SLOT DirectSQLStatementPrivate:Descriptor>>;
    set::<
      (#
      <<SLOT DirectSQLStatementSet:DoPart>>
      #);
    getExpanded:
    (* Get the contents of this(directSQLStatement)
     * as it would appear if the statement was executed now *)
      (# value: ^text
      <<SLOT directSQLStatementGetExpanded:DoPart>>
      exit value[]
      #)
  do INNER
  #);
preparedSQLStatement:
(* Use this statement type if a statement will be executed multiple
 * times with different bindings.
 * ONLY IMPLEMENTED AS A DIRECT STATEMENT
 *) directSQLStatement
```

```
  (# <<SLOT PreparedSQLStatementLib:Attributes>> do INNER #);
resultSet:
(* A result of an SQLStatement.
 * If columnCount <> 0 then the resultSet is scanable.
 * A resultSet can be scanned at most once.
 *)
  (#
     <<SLOT ResultSetLib:Attributes>>;
     columnCount:
     (* The number of columns in this(resultSet) *) integerValue
       (#  <<SLOT ResultSetInfoNoOfCols:DoPart>> #);
     rowCount:
     (* If the statement that created this(resultSet) was a
      * 1. INSERT, DELETE or UPDATE statement
      *     rowCount yield the number of rows
      *     affected by the statement
      * 2. SELECT statement
      *     rowCount MAY (i.e. does not in all circumstances) yield
      *     the number of rows selected
      *) integerValue (#  <<SLOT ResultSetInfoRowCount:DoPart>> #);
     column:
     (* A column in this(resultSet) *)
       (#
          name: ^Text;
          no: @Integer;
          dataType:
          (* The BETA pattern corresponding to the SQL datatype
           * for this column. If the SQL datatype is DATE, TIME or
           * TIMESTAMP then the corresponding BETA pattern will
           * be time
           *) ##Object;
          dataTypeName: (* DBMS specific type name *) ^Text;
          dataTypeNo: (* ODBC specific numbering of SQL datatypes *)
            @integer;
          nullAble: @Boolean
       #);
     getColumn: (* Gets the column number 'i' in this(resultSet) *)
       (# i: @Integer; res: ^column
       enter i
       <<SLOT ResultSetInfoGetColumnInfo:DoPart>>
       exit res[]
       #);
     getColumnByName:
     (* Gets the column designated by 'name' in this(resultSet) *)
       (# name: ^Text; res: ^Column; nameNotFound:< exception
       enter name[]
       <<SLOT ResultSetInfogetColumnInfoByName:DoPart>>
       exit res[]
       #);
     result:
     (* A row in this(resultSet). If the enter parameter of the
      * scan operation that created this result contained
      * %_ placeholders the values of these may be retrieved
      * by using the markers below
      *)
       (#
          <<SLOT ResultLib:Attributes>>;
          marker:
          (* Gets the value of a %_ placeholder and advances
           * the placeholder currently referred to
           *) (#  <<SLOT resultSetScanMarker:DoPart>> #);
          b: marker
```

```
      (# value: @boolean
      <<SLOT resultSetScanB:DoPart>>
      exit value
      #);
    c: marker
      (# value: @time
      <<SLOT resultSetScanC:DoPart>>
      exit value
      #);
    d: marker
      (# value: @time
      <<SLOT resultSetScanD:DoPart>>
      exit value
      #);
    f: marker
      (# value: @real
      <<SLOT resultSetScanR:DoPart>>
      exit value
      #);
    i: marker
      (# value: @integer
      <<SLOT resultSetScanI:DoPart>>
      exit value
      #);
    s: marker
      (# value: ^text
      <<SLOT resultSetScanS:DoPart>>
      exit value[]
      #);
    t: marker
      (# value: @time
      <<SLOT resultSetScanT:DoPart>>
      exit value
      #);
    private: @<<SLOT resultSetResultPrivate:Descriptor>>
  #);
scan:
(* Scans over the rows of this(resultSet).
 * A pattern may be entered. The pattern entered may either
 * contain
 * 1. Named columns of the form
 *          'name1:var1 name2:%i name3:var2'
 *     which means that the column named 'name1'('name3')
 *     is bound to the variable named 'var1'('var2') and the
 *     column named 'name2' may be retrieved from the
 *     current result using the 'i'-marker.
 * or
 * 2. Consecutive columns of the form
 *          'var1 %i var2'
 *     which means that the first (third) column in each result
 *     is bound to the variable named 'var1'('var3') and that
 *     the value of the second column may be retrieved from the
 *     current result using the 'i'-marker
 * or
 * 3. Nothing
 *     which means that the values of the column may be
 *     retrieved from the current result using the %_ markers
 *     corresponding to the types of the values in the result
 *)
  (#
    current: @result;
    varNotDeclared:<
```

```
                (* Raised if variable used in the entered
                 * pattern was not found *) exception
                  (# name: ^text enter name[] do INNER #);
                unknownColumn:<
                (* Raised if this is a named scan and
                 * named column was not found *) exception
                  (# name: ^text enter name[] do INNER #);
                pattern: ^text
             enter pattern[]
             <<SLOT resultSetScan:DoPart>>
             #);
           resultSetException:< BetaDBCException
             (#
             do INNER resultSetException
             #);
           resultSetWarning:< BetaDBCWarning
             (#  do INNER resultSetWarning #);
           private:
             @<<SLOT ResultSetPrivate:Descriptor>>
        #);
      open:<
      (* Opens this(connection). The name of the connection
       * to be opened must be supplied.
       * Supplying userName and/or password is voluntary.
       *)
        (#
           name: ^text;
           userName: ^text;
           password: ^text;
           openConnectionException:< BetaDBCException
             (#  do INNER openConnectionException #);
           openConnectionWarning:< BetaDBCWarning
             (#  do INNER openConnectionWarning #)
        enter (name[],userName[],password[])
        <<SLOT ConnectionOpen:DoPart>>
        #);
      close:<
        (#
           closeException:< BetaDBCException
             (#  do INNER closeException #);
           closeWarning:< BetaDBCWarning
             (#  do INNER closeWarning #)
        <<SLOT ConnectionClose:DoPart>>
        #);
      connectionException:< BetaDBCException
        (#  do INNER connectionException #);
      connectionWarning:< BetaDBCWarning (#  do INNER connectionWarning #);
      private: @<<SLOT ConnectionPrivate:Descriptor>>
  #);
BetaDBCException:
(* Low level interface for catching exceptions.
 * A general exception message is supplied in msg,
 * SQL states and native error codes in SQLState,
 * NativeError in a comma-separated list.
 *) Exception
  (#
     SQLState: @text;
     NativeError: @text;
     HandleType: @integer;
     Handle: @integer
  enter (HandleType,Handle)
  <<SLOT DBEnvBDBCException:DoPart>>
```

```
  #);
BetaDBCWarning: BetaDBCException
  (#  do true->continue; INNER #)
```

6 Text files for the tutorial

6.1 *createmoviedbtables.txt*

```
CREATE TABLE MovieStar (
    name CHAR(30),
    address VARCHAR(255),
    gender CHAR(1),
    birthdate INTEGER
);

CREATE TABLE Movie (
    title VARCHAR(255),
    year INTEGER,
    length INTEGER,
    inColor BIT,
    studioName CHAR(50),
    producerCNo INTEGER
);

CREATE TABLE StarsIn (
    movieTitle VARCHAR(255),
    movieYear INTEGER,
    starName CHAR(30)
);

CREATE TABLE MovieExec (
    name CHAR(30),
    address VARCHAR(255),
    certNo INTEGER,
    netWorth INTEGER
);

CREATE TABLE Studio (
    name CHAR(50),
    address VARCHAR(255),
    presCNo INTEGER
);
```

6.2 *deletemoviedbtables.txt*

```
DROP TABLE MovieStar;

DROP TABLE Movie;

DROP TABLE StarsIn;

DROP TABLE MovieExec;

DROP TABLE Studio;
```

6.3 *populatemoviedbtables.txt*

7 References

[Date 1993] Date, C.J. and Darwen, H., *A Guider to the SQL Standard,* Addison Wesley, Reading, MA, 1993.

[Geiger 1995] Geiger, K. *Inside ODBC,* Microsoft Press, 1995.

[Ullman 1997] Ullman, J.D., Widom, J., *A First Course in Database Systems*, Prentice Hall International, 1997.